# Response to reviewers' comments for Paper # 278

We are deeply grateful to all the reviewers for the generous and insightful suggestions which help us to improve this work to a better level. In the following, we provide explicit response to each raised concern and address our corresponding revisions in the new submission.

## 1. RESPONSE TO REVIEWER 1

**1.1** *The main con that I've spotted is the fact that the algorithm might have been more nicely framed also within the Spark environment by taking advantage of the various possibilities offered by it (e.g. caching of RDDs)*

**Response:** We agree with the reviewer that utilizing advanced Spark features would benefit our solutions. Indeed, our SPARE algorithm has already taken advantages of the Spark-only features: (1) caching of RDDs and (2) DAG execution engine, to achieve speedups.

As described in Section 5.3, our SPARE algorithm uses a run-time best-fit strategy to achieve load balance. The best-fit strategy allocates the largest unassigned star to the most empty reducer, which balances the workloads for each reducer. Since the strategy needs to know the sizes of all stars, it takes place after the Map phase and before the Reduce phase. Leveraging Spark's DAG execution engine, we can easily augment traditional Map-Reduce workflow to the Map-Planning-Reduce workflow. Furthermore, we utilize the RDD caching feature of Spark to store the map result (i.e., RDDs of stars) during the planning phase and reuse them in the reduce phase.

In Section 6.2.2 Figure 8, we validate the benefits of adopting these Spark features by comparing SPARE with SPARE-RD. SPARE-RD is identical to SPARE except it uses random allocation rather than best-fit strategy. Benefit from the two Spark features, the SPARE the best-fit strategy only takes around 4% the total execution time. In return, SPARE gains 15% speedup compared to SPARE-RD.

We do not consider utilizing other Spark extensions such as Spark-GraphX, Spark-Streaming and Spark-MLlib because these extensions, as their name suggest, are for different application scenarios.

## 2. RESPONSE TO REVIEW 2

**2.1** *The GCMP generalization is not particularly novel. Putting a maximum gap size on consecutive segments is well-known in sequence mining published more than 10 years ago.*

**Response:** We admit that the gap parameter is also used in mining Gap-restricted Sequential Patterns (GSP). However, our work is the first to introduce the gap constraint in co-movement mining in trajectory domain, and we are the first to resolve the loose-connection anomaly (as shown in Figure 2). Moreover, we provide a unified model to capture existing pattern models which is also not proposed before.

Besides, GSP and GCMP are designed with different goals. Thus it is infeasible to model GCMP using GSP. In particular, the goal of GSP is to find the frequently occurred subsequences which conform to the gap constraint. This definition ignores the relationships among the involved objects. As a consequence, the mined GSP results cannot guarantee the spatial closeness of the involved objects, which contradicts with *closeness* requirement of GCMP (Section 3 Definition 2). For example, in the sample trajectory database in Figure 1, the most frequent temporal sequence for GSP is $(1, 2, 3, 4, 5, 6)$ because it appears in every trajectory. However, the involved objects $\{o_1, o_2, o_3, o_4, o_5, o_6\}$ are NOT all spatially close at any of the timestamps thus these objects do not form a proper GCMP. We summarize this difference in Section 2.5.

**2.2** *In fact, I have doubts about formulating the GCMP patterns as proposed. Are we really interested in all sets of movements beyond a cardinality of size $M$? Take the Taxi dataset as an example. Let say that there are lots of taxis going from the airport to downtown. Let say that there are 1000 such taxis. For a given $M$, are we interested in $\binom{1000}{M}$ answers? So this speaks to the problem of picking $M$. If $M$ is 500, what is $\binom{1000}{M}$? In fact, even if the system gives the single answer of $\binom{1000}{1000}$, I am not sure I am interested in this pattern as I already know that there are many taxis going from the airport to downtown. What I think I am really interested in are GCMP that are "anomalous", which is much harder to define.*

**Response:** We appreciate the reviewer's concerns on the size of the output. In fact, both TRPM and SPARE have considered compressing the output by only discovering the patterns with larger object set. Particularly, in the Line Sweeping Mining (Algorithm 1) of TRPM, whenever a pattern $c$ becomes valid, it is directly outputted (Lines 14-16).

This prohibits outputting $c$'s subset. Similarly, in the Apriori Enumerator (Algorithm 3) of SPARE, a pattern $c$ is outputted if none of its supersets could become a valid pattern (Line 12-14). This ensures the output $c$ is not a subset of some pattern. These mechanisms effectively condense the output by subsuming smaller patterns using their supersets. Linking to the Taxi example, if the ground truth contains 1000 taxis travel together, then both TRPM and SPARE tend to output the single pattern. In addition, real detection of "anomalous" co-moment pattern needs to impose stringent constraints via $M, K, L, G$. This would further reduces the size of the output.

**2.3** *Regarding the second weak points, one line of related work is the superimposition of constraints on spatiotemporal mining. An example is a road network. In other words, given a road network, the network imposes constraints on GCMP.*

**Response:** We understand that spatiotemporal mining with domain-specific constraints may look similar to GCMP. However, we are unaware of existing works on the same GCMP model with domain-specific constraints.

Generally, patterns defined with domain-specific constraints are hard to be generalized. This is because constraints available in one domain (e.g., road direction in road networks) are often not available in other domains (e.g., no road directions available for visitor movements in a shopping mall). Nevertheless, when deploying on a specific domain, our GCMP solutions are amendable to incorporate more domain-specific constraints (i.e., rewrite the `sim` operation in Algorithm 3). We summarize the discussion in Section 2.3.

# 3. RESPONSES TO REVIEWER 3

**3.1:** *Though the implementation references Spark as the implementation platform (as it also reflects in github repo provided), the algorithm design is mostly limited to MapReduce, aka only Hadoop, which is a very small subset of Spark. This may have a negative impact on the baseline implementation. Particularly, recent releases of Spark have introduced window functions that can be applied directly in the sliding window scenario here. Certainly, the algorithm has to be redesigned to use DataFrame (and/or Spark SQL) interface, it has been noted that this is a very efficient way to execute window functions in Spark.*

**Response:** We thank the reviewer for suggesting a better way to implement the baseline algorithm (i.e., TRPM). With window functions, TRPM could be rewritten as in the following pseudocode:

```
WindowSpec ws := Window.orderBy('time')
                       .rowBetween(0, η);
Pattern pt := LineSweep('clusters').over(ws);
```

However, the algorithm is challenging to be implemented after our investigations. The major reason is that, in current version of Spark-SQL [1], User Defined Aggregate Function (UDAF) on window operator is not supported [2]. Without UDAF, we cannot attach the `over` keyword to the Line Sweep algorithm. A workaround is to implement the Line Sweep using only Spark-SQL primitive aggregate functions (e.g., sum, avg, rank and nth-tile). But it is beyond our abilities.

On the other hand, we believe the performance boost on TRPM from Spark-SQL would be very limited. The reason is that our Line Sweep algorithm (Algorithm 1) is distributive. That is the result over an input cannot be constructed from the result over a subset of the input. This indicates that no partial results could be shared among all windows. In such a case, Spark-SQL has to process each window independently in parallel, which is equivalent to our TRPM implementation.

**3.2:** *Most implementations on Hadoop and Spark are very sensitive to data partitions, i.e. prone to data skewing issues. It seems that the (starpartition) implementation does not take this into account, and only use the default partitioning. It would be very helpful to have a discussion on this topic.*

**Response:** We study the skewness by examining the load balance of each algorithm. We revise Section 6.2.2 by comparing SPARE with different data partition strategies. In particular, we compare SPARE which contains a best-fit strategy (Section 5.3) and SPARE-RD which adopts the default random partition strategy. We present the summary of the execution time for each method on the longest job, the shortest job and standard deviation of all jobs in Table 9. As the table shows, SPARE has a better load balance (low deviation) than SPARE-RD which indicates the effectiveness of the best-fit strategy.

**3.3:** *In particular, it would be great to provide the difference in the number of partitions/splits, the amount of processing and memory usage (i.e., vcore and memory seconds) between TRPM and SPARE*

**Response:** In our implementation, we take a fixed 486 partitions (Section 6) for each RDD. The partition number is determined according to the parallel tuning guide by Cloudera [3]: there are $3 \times 54 = 162$ cores and each core is able to processes 3 partitions [4] by multithreading.

Next, we add Table 7 (Section 6.1) to compare the resource usage (e.g., Vcore-seconds, Memory and Running Time) between SPARE and TRPM. The table tells that both SPARE and TRPM are resource efficient. In particular, the RDDs for both SPARE and TRPM only take less than 20% of the available memory. This infers the potential of SPARE and TRPM in handling even larger trajectories with our current cluster resources.

**3.4:** *A plot that breaks down the performance gain by each method would be greatly appreciated by the readers.*

**Response:** We include the analysis of the breakdown cost of SPARE by taking TRPM as a reference in Figure 8. SPARE and TRPM algorithms are of the similar structure: partitioning (Star partition v.s. $\eta$-replicate partition) is in the map-shuffle phase, and mining (Apriori Enumeration v.s. Line Sweep) is in the reduce phase. In our experiment, we treat the execution time for each phase as the cost of the corresponding methods.

As shown in Figure 8, both star partition and apriori enumeration contribute to the final performance gain of SPARE.

[1] Both the stable Spark 1.5.2 and the newly released Spark 1.6.2

[2] JIRA Spakr-8641: Native Spark Window Functions `https://issues.apache.org/jira/browse/SPARK-8641`

[3] Spark Performance Tuning `http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-1/`

[4] This number may differ for other CPU models, and can be determined empirically.

In particular, in the Map-Shuffle phase, SPARE saves 56%-73% in time as compared to TRPM. This indicates that less data are transformed and shuffled in the Star partition. In the Reduce phase, SPARE saves 46%-81% time as compared to TRPM. This confirms the efficiency of our Apriori enumerator.

**3.5** *Some choices of words may need to be reconsidered: for example, "a bunch of" might not be appropriate in a technical paper.*

**Response:** We have changed these unprofessional terms.

**3.6** *References to star partitioning and apriori pruning are missing. Though these are well known, they need to be clearly cited.*

**Response:** We add the corresponding references in Section 5.1 and Section 5.2 when we describing our algorithms.

**3.7** *In "In contrast, when utilizing the multicore environment, SPAREP achieves 7 times speedup and SPARES achieves 10 times speedup.", was "multicore" referring to the use of all 16 cores in one of your node? The specification of the machine was not clear.*

**Response:** We use all cores in the single node (i.e., 4 executors each with 3 cores) to conduct the experiment. We revise the description in Section 6.2.3.

**3.8** *The computation of "eta" was slightly different than that in the paper*

**Response:** We have updated the GitHub repository to rectify the typo in the equation.

# 4. RESPONSE TO META REVIEWER

**M.1** *Novelty: The GCMP generalization is not particularly novel. Please elaborate on the novelty of the work.*

**Response:** In addition to our response to comments 2.1, another novelty of our work lies in the techniques. Although the idea of star partition and apriori enumeration are well-known, linking them together to solve the problem in trajectory domain has not been attempted before. Besides, we input heavy details on these algorithms (e.g., theoretical bounds of partition and anti-monotonicity) which have not been previously proposed yet.

**M.2** *Parameter setting: Are we really interested in all sets of movements beyond a cardinality of size M? Please elaborate on the motivation.*

**Response:** Addressed in the response for comments 2.2.

**M.3** *Spark implementation: the paper references Spark as the implementation platform but the algorithm is limited to MapReduce. Spark has capabilities beyond MapReduce such as window functions. Please provide an implementation that uses the relevant features of Spark, or a convincing discussion of how these features can be useful and why they were not used.*

**Response:** Addressed in the response for comments 1.1 and 3.1.

**M.4** *More details in the performance evaluation: Please provide more details about data partitioning and the effect of skew. Also provide details about how star partitioning and a priori pruning contribute to performance. Please provide references to these two methods.*

**Response:** Addressed in the response for comments 3.2, 3.3, 3.4 and 3.6.